Microelectronics Conference VLSI 1987
Melbourne, 8-10 April, 1987

# A Fast VLSI Implementation of a FIFO Queue

D. BERRY, A. HEADLAM, R.K. LOANE, J. PARRY, T.K. WANG
(Honours) Students, University of Tasmania
A.H.J. SALE
Professor of Information Science, University of Tasmania

**SUMMARY** The paper describes a hardware implementation in nMOS of a first-in, first-out (FIFO) queue. The implementation has independently operating insertion and extraction logic which is capable of achieving high speeds of less than 200*ns* per operation, and may be entirely contained on a single chip. A regular cellular structure is described which is capable of extension both in the direction of wider queued items and in the direction of maximum queue size. The implementation was carried out at the University of Tasmania by the first five authors under the supervision of the last-named author.

## 1 CONCEPT

There is an increasing trend for algorithms to be implemented in hardware as integrated circuit technology achieves higher and higher densities of active components and is thus capable of implementing more complex structures. Generally the best implementations are derived from structures with a high degree of regularity. The implementation described in this paper is that of a first-in, first-out (FIFO) queue in silicon. The design is based on a regular array of cells each of which communicates only with its near neighbours.

A integrated circuit implementation of a queue is capable of higher speeds than can be obtained using a conventional microprocessor and memory programmed to implement a queue. High speed queueing can be useful in some applications, for example in chemical experiments and satellite communications. In some of these cases the data arrives at a very high burst rate; the consuming process may be capable of matching the average arrival rate but not able to handle the peak arrival rate during the burst. The reverse situation is also possible. Buffered high-speed channels (or *pipes* in *Unix* parlance) between processors are also useful simply to smooth out production and consumption rates betweeen different processes.

## 2 STRUCTURE

The FIFO queue described is based on a concept presented by Al-Khalili & Ali (1986). This concept itself is derived from earlier work on systolic queues by Leiserson (1979). The initial stages of our design closely parallels a description in the paper by Al-Khalili & Ali and is presented below. It is emphasized that this structure is described at a conceptual level; there is scope for implementation variations.

The queue is composed of two extended vectors of cells $L_i$ and $R_i$ (Left and Right), where i = 1 to M. Each cell may hold a single item of queued data. Besides the queue data each cell has a full/empty status denoted by $SL_i$ and $SR_i$ for the left and right cells respectively. Each cell $L_i$ is capable of transferring data to the cell $L_{i+1}$ or to the cell $R_i$. Each cell $R_i$ is capable of transferring data to the cell $R_{i-1}$. This cellular interconnection structure is shown in Figure 1.

In our design we further postulate the existence of a synchronous clock signal distributed to all cells, and assume that all cells assume a new state and data at the clocked instant, dependent on their previous state and that of their immediate neighbours. If data is transferred out of a cell it assumes the empty state; unless specified otherwise below a cell retains its previous state and data (if full).

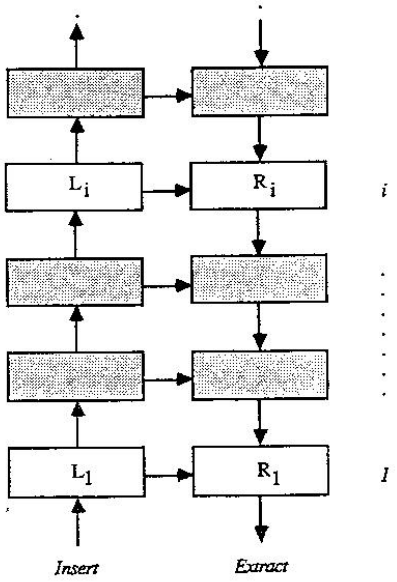


**Figure 1. Interconnection of queue cells.**

If a full right cell $R_i$ finds an empty neighbour cell below it ($R_{i-1}$) then the data moves downwards on the next clock cycle. The extraction of an item from the front of the R vector makes the head cell empty. On successive clock cycles an empty cell (a *hole*) moves up through the full part of the vector until it reaches the tail of the queued items in the R vector. Since items cannot be extracted from the head of the queue on successive clock cycles (see below for discussion of peak extraction rates), each hole is surrounded by full cells as it moves up the R vector of queued items.

If a full left cell $L_i$ finds itself above the tail of the right queue then its data moves across from the left cell $L_i$ to its corresponding right cell $R_i$. From the property deduced above, the tail of the R vector is marked by both $R_i$ and $R_{i+1}$ having the empty status. However, if this vector transfer cannot be carried out then a full left cell $L_i$ will move to occupy an empty left neighbour cell $L_{i+1}$ on the next clock cycle, moving up in the L vector.

Consider first the case where the number of queued items N is less than M. With this organization a queue from which no extractions have been made for some time will be in a steady state with the right vector holding queued items from $R_1$ to $R_N$. This situation is shown in Figure 2. If an extraction is made, the head item is moved and the vacancy moves up the right vector (like a hole in semiconductors) until it reaches the tail and merges with the rest of the empty cells at that point. If an insertion is made, the resulting full left cell moves up the left vector until it reaches an index value just above the last full cell of the right vector, and then it transfers across to the right vector.

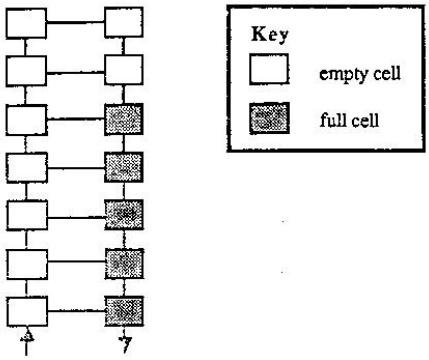


Figure 2. Steady state following no terminal activity for an extended period.

The maximum sustainable extraction rate is achieved if an item is extracted every two clock cycles. Data cannot be extracted faster than this, for if data is removed from cell $R_1$ on clock cycle $\tau$, then on cycle $\tau+1$ the cell is empty waiting for transfer from $L_1$ or $R_2$. If this occurs then the data can be removed on cycle $\tau+2$. In an infinitely large system, the R vector would be populated by alternate full and empty cells all moving synchronously in the direction of smaller indexes (downwards).

Since the maximum transfer rate of data up the L vector is also achieved under the same conditions, the maximum insertion rate is the same: one insertion every two clock cycles. This rate can be sustained under conditions of minimal buffering as well as maximal buffering, for data takes only two cycles to transfer from the input to the output if the queue is empty. A snapshot of the resulting steady state is shown in Figure 3.

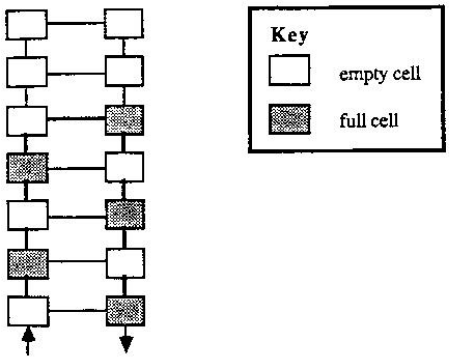


**Figure 3. Steady state at maximum traffic rates.**

In the above case the non-infinite extent of the vectors can be ignored since no data is able to reach the end of the vectors. However, consider also the case where the number of queued items is greater than M. In this case it is necessary that appropriate end conditions are provided to the last cells $L_M$ and $R_M$. The cell $L_M$ must be unable to transfer its data upwards under any conditions and thus it must be provided with boundary conditions corresponding to a full cell $L_{M+1}$. The right cell $R_M$ must appear to be past the tail of the right vector if it is empty, so the simulated cell $R_{M+1}$ above it must appear to be empty. (Due to a variation in the design, the implementation has $L_{M+1} = R_{M+1} = R_M$.)

Under these conditions and light load all cells may be filled giving the queue a maximum capacity of 2M items. However, if maximum data transfer rates are being achieved then half of the cells must be empty, so the maximum queue storage under heavy load is M items.

The boundary conditions at the external end of the queue are also easy to derive. Data may be inserted if $L_1$ is empty. Data may be extracted if $R_1$ is full.

Sometimes there is a necessity to determine whether the queue is empty (no stored items) or full (no more items can be accepted unless an extraction takes place). An empty queue may be signalled if all of the cells $L_1$, $L_2$, $R_1$ and $R_2$ are empty, for under these conditions no data is in transit to the output. A full queue cannot be externally detected except by observation of the input cell $L_1$. If it remains full for 2M cycles, the queue is full. This caters for the case of an empty 'hole' travelling through both vectors from the output to the input. Fortunately a requirement to determine whether a queue is full is not often required and permission to insert is usually sufficient.

The properties mentioned above can be proved from an analysis of the states of the queue, but this proof is not presented in this paper as it is oriented to the description of a particular implementation.

*Deviations from the previously described queue*

The description of the queue structure presented here generally follows the presentation in Al-Khalili & Ali (1986) but has a few deviations. In particular their description assumes that extraction of data requires two active clock cycles. On the first the queue is interrogated to see if data will be available on the next clock cycle. This is possible if any of the cells $R_1$, $L_1$ or $R_2$ are full. In the first case data is waiting at the head of the queue and in the second and third cases it will be transferred there during this clock cycle. Data is then removed on the second clock cycle.

Their description also requires two active clock cycles for insertion. In this case the queue is interrogated to see if the insertion cell will be empty on the next clock cycle. This is assured if $L_1$ is empty (since it cannot be filled in this cycle) or if $L_2$ is empty. If there is data in $L_1$ it may move across to $R_1$ or move up to $L_2$ during this cycle.

There are also two errors in the paper. It states '*if data cannot be pushed the queue is completely full; if data cannot be popped, it is empty.*' The first of these two assertions is not correct. It is possible for the queue to have been completely filled and for some extractions to have taken place subsequently so that it is no longer full. However it will take a time of the order of 2M cycles for a hole to travel back through the queue to the start and only then will it be possible to insert more items. Detection of a completely full status thus requires a global view of the structure or alternatively a finite-state machine which monitors events at the input and output.

The second error requires analysis of the operation. (A paper by A.H.J.Sale is in preparation proving properties of the systolic array.) Al-Khalili & Ali assume that the queue never has two consecutive holes below the tail on the right; however the description provided allows situations similar to that shown in Figure 4 to occur. The order of items C and D are reversed on extraction. This error was resolved by allowing $L_{i-1}$ to

move to $R_i$ under the condition that $L_i$, $R_i$ and $R_{i+1}$ are all empty.

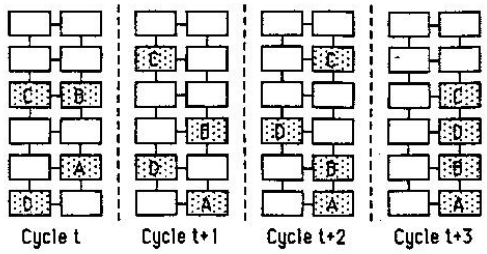


**Figure 4. Illustrating loss of ordering.**

## 3 BASIC DESIGN LAYOUT

The design of the queue was based on three major components. If the FIFO queue is required to buffer N-bit data words, and to have a total capacity of 2M words, the requirements for each of the components are:

- The interface cell, which takes the *pop* and *push* signals from the outside world and provides the bottom control cell with the correct values for $SL_0$ and $SR_0$. It also takes the outputs $SL_1$ and $SR_1$ from the bottom control cell and provides two signals to the outside world, *push done* and *data available*. Only one of these cells is required.

- The control component, which consists of M control cells, each of which remembers the values $SL_i$ and $SR_i$, takes the four signals $SL_{i-1}$, $SL_{i+1}$, $SR_{i-1}$ and $SR_{i+1}$ from the neighbouring control cells and provides the signals:
  - 'move $L_{i-1}$ into $L_i$',
  - 'move $L_i$ into $R_i$', and
  - 'move $R_{i+1}$ into $R_i$'.

  Note that resolving the double hole problem allows both the signals 'move $L_{i-1}$ into $L_i$' and 'move $L_i$ into $R_i$' to be simultaneously present. All the cells are identical except for a slight modification to the bottom cell.

- The memory component, which consists of an $M \times N$ array of memory cells. Each of these cells holds both the $L_i$ and $R_i$ bits in dynamic memory. Five transmission gate paths are required in each cell; three controlled by the three signals from the control cell together with two data refresh paths.

A layout of cells to implement the queue is shown in Figure 5.

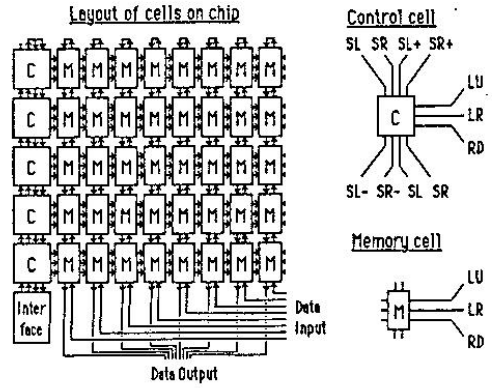


**Figure 5. Layout of cells on chip**

Once the logical interface between the cells was decided, a program was written to simulate the logical design of the array, in order to fully test the logical design of the chip. At this stage one of the previously described design errors was detected: that illustrated in Figure 4 which allows the order of queued items to be lost due to the introduction of 'double holes'.

## 4 DESIGN CONSIDERATIONS

### 4.1 Overall Design

As the authors already had experience in designing chips using nMOS technology, and were still gaining prototyping experience with CMOS designs, it was decided to use nMOS. It was also decided that the $\lambda = 2.5\mu m$ AWA fabrication facility would be used through the University of New South Wales MPC service.

There was another major decision at the start of the project. It was possible to preserve the logical structure of the queue as described earlier, where all the left bits are on one side of the chip and all the right bits on the other. The other possibility, which was chosen in the final design, is to interleave the left and right bits as was shown in Figure 5. A separated design would have required the two sides to be connected at each row, resulting in a river crossing the array, requiring one wire for each bit of data width and thus resulting in a waste of space for the N wires and their separations.

To simplify the design and avoid race conditions developing in the control cell, a two phase globally clocked design was chosen. Phase one ($\phi_1$) would be used for the memory cell refresh and next-state computations while phase two ($\phi_2$) would be used to change the memory cell values and the corresponding empty/full bits in the control cells.

### 4.2 Memory Cell Design

When designing this cell, a number of decisions had to be made:

- Would static or dynamic storage be used?

- How would the layout of the memory cell fit with the layout of the control cell, which would be very much larger in area?

- If dynamic storage was used, how would it be refreshed?

Due to a significantly smaller requirement for chip area per cell bit, dynamic storage was chosen, and a global clock signal was used to refresh it. When considering the cell layout, it was decided to use a layout which had a reasonably long edge facing the control cell with the other edge being very short (a cell which was deep in the propagation direction but narrow transversely). This resulted in a cell design where the three data paths were stacked on each other, with wires running in parallel as shown in Figure 5.

When considering the design layout, it was found that it was possible to economize on cell area by inverting the data bits every time they are transfered from one cell to the other. The fact that the bits are inverted on every move is not a problem as the data words will always be moved an even number of times before they emerge at the far end of the queue. For every left side cell the data enters, it must enter exactly one right side cell.

### 4.3 Control Cell Design

Two different logical designs were suggested for the control cell. One required all the data coming *into* a cell pair $L_i$ and $R_i$ to be controlled by a single control cell (which we named the Z design) and a second where the control cell controls the paths *between the i–1 and i pairs* (which we named the U design). The U design has the added complexity that the cell which controls new data coming into a memory bit is not always the control cell which remembers if there is valid data in the memory bit. By a majority decision the Z design was chosen, and no further explorations of the U design were undertaken.

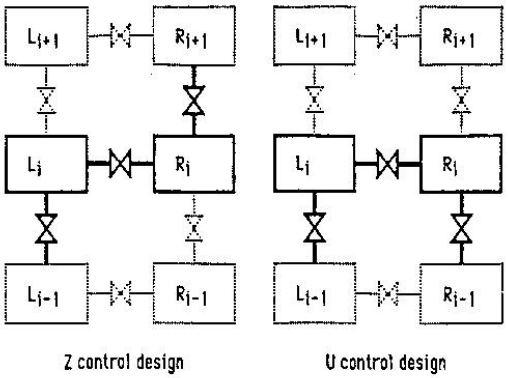


**Figure 6. Alternative control cell paths**

Having decided to use the Z design for the control cell, the next decision was whether to use discrete custom logic designed by hand or whether to use a programmed logic array (PLA). Due to time limitations constraining the students, it was decided to use a PLA, and the following equations were used to produce the single control cell. (Note: All values default to false if not specified.)

```
if R_{i+1} ∧ ~R_i      then /* Move right bit down */
                            {RD := true; NextR := true}

if L_i ∧ ~R_i ∧ ~R_{i+1}
                       then /* Move left bit right */
                            {LR := true; NextR := true}

if ~L_i ∧ L_{i-1} ∧ ~R_i ∧ ~R_{i+1} ∧ ~R_{i-1}
                       then /* Move lower left bit up and right */
                            {LU := true; LR := true; NextR := true}

if ~L_i ∧ L_{i-1} ∧ (R_i ∨ R_{i-1}) ∧ ~(~R_i ∧ ~R_{i+1} ∧ R_{i-1})
                       then /* Move lower left bit up */
                            {LU := true; NextL := true;}

if R_i ∧ R_{i-1}       then /* Can't move the current right bit */
                            {NextR := true;}

if L_i ∧ L_{i+1}       then /* Can't move the current left bit */
                            {NextL := true;}
```

With the bottom control cell, the third and fourth equations must be changed as the data is required to enter the chip and not be transferred right by a prior but non-existent control cell. These equations become:

```
if ~L_i ∧ L_{i-1} ∧ ~R_i ∧ ~R_{i+1}
                       then /* Move lower left bit up and right */
                            {LU := true; LR := true; NextR := true}

if ~L_i ∧ L_{i-1} ∧ ~(~R_i ∧ ~R_{i+1})
                       then /* Move lower left bit up */
                            {LU := true; NextL := true;}
```

The sizes (1 λ = 2.5 μm) of the final primitive cells are:

*Memory cell*
80 × 53 λ
custom design
4 pull-up transistors and 9 enhancement transistors

*Control cell*
160 × 200 λ
PLA design

For estimating purposes, a single row of an 8-bit queue would occupy a chip area of:

$$160 \times (200 + 8 \times 53)\lambda = 400\mu m \times 624\mu m.$$

A 6mm chip could contain 9 columns each of 15 rows, resulting in a queue capacity of 2 × 9 × 15 = 270 bytes. With optimization of cell design, this could easily be doubled to say 500 bytes per chip.

## 5 Testing Procedure

Simulation of the chip was carried out using a bottom up approach where the primitive cell structure was simulated using a special purpose program, then the major components were simulated using a commercial electrical network simulator, followed by a few simulations of the full chip.

The computer time required to simulate a primitive cell was small, taking less than a minute for a single memory cell and a couple of minutes for the control cell. However, when simulating the larger components this time increased by a factor of between 10 and 20, with a simulation of the full chip taking about an hour to complete on the Department of Information Science's VAX-11/750.

The computer simulations were run at a simulated rate of 5 MHz with a two phase clock with reasonable gaps between the two phases where both signals were low. At this rate it would be possible to insert and extract data at the rate of one item every 200*ns*. With care it may be possible to approach independent peak insertion and extraction rates of one item each 100*ns*. At the date of writing a fabricated chip had not been received for testing and verification of these estimates.

## 6 Applications

The queue described is a hardware implementation of a FIFO queue. It will probably not be cost competitive against a dedicated conventional microprocessor and memory with a software implementation of a FIFO queue, where this can be used.

The applications envisaged for this design are:

- Very high speed queueing, where the peak data arrival rates or peak data extraction rates exceed that which can be achieved by a programmed loop (say to100 kwords/sec) or direct memory access techniques (say to 1 Mwords/sec). Such situations may arise in commmunications circuits, such as satellites, and in high-speed message-passing links between concurrent distributed multiprocessors such as a transputer (INMOS, 1985) network.
- On-board a chip, where a small but high speed queue is required to decouple two independently operating devices producing and consuming data.

## 7 Further Work

A CMOS implementation of the queue is planned, together with an optimization of the chip area occupied by each row of the queue. A paper analysing the theory of the FIFO queue, together with similar derived structures, is in preparation.

## Acknowledgments

This paper describes a collaborative design effort by students enrolled in the Fourth Year (Honours) VLSI Design unit offered by the Department of Information Science at the University of Tasmania, under the supervision of Professor A.H.J.Sale. The work was carried out with the assistance of the following grants:

- Australian Research Grant F8516057I,
- a University of Tasmania Research Grant, and
- an Australian Computer Research Board Grant.

The comments of members of the University's *Silicon Design Group* and members of staff of the Department of Information Science are gratefully acknowledged, together with the assistance given by *Integrated Silicon Design Ltd* and the University of New South Wales.

## REFERENCES

Al-Khalili A.J. & Ali Z. (1986). A Fast Systolic FIFO Queue. *VLSI Systems Design*, May 1986, pp76-80.

Leiserson, C.E. (1982). *Area-Efficient VLSI Computation.* MIT Press, Cambridge, MA.

INMOS (1985). *Transputer reference manual.* INMOS Ltd, Bristol, UK.